

**Aufgabe 1.1: Einfachauswahl-Fragen (22 Punkte)**

Bei den Multiple-Choice-Fragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Multiple-Choice-Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (  ) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten!

a) Welche Aussage über `fork()` ist richtig?

2 Punkte

- Dem Eltern-Prozess wird die Prozess-ID des Kindes zurückgeliefert.
- Der Aufruf von `fork()` ersetzt das im aktuellen Prozess laufende Programm durch das als Parameter angegebene Programm.
- Der Kind-Prozess bekommt die Prozess-ID des Elternprozesses zurückgegeben.
- `fork()` bekommt als Parameter einen Zeiger auf eine Funktion übergeben, die nach dem Aufruf in einem neuen Thread gestartet wird.

b) Welche Aussage zum Thema Adressräume ist richtig?

2 Punkte

- Im realen Adressraum sind alle theoretisch möglichen Adressen auch gültig.
- Der Zugriff auf eine virtuelle Adresse, die zum Zeitpunkt des Zugriffs nicht im Hauptspeicher abgebildet ist, führt zu einem Trap.
- Die Größe eines virtuellen Adressraums darf die Größe des vorhandenen Hauptspeichers nicht überschreiten.
- Bei Seitennummerierung besitzt jede Seite eine unterschiedliche Größe.

c) Welche Aussage zum Thema Speicherzuteilung ist richtig?

2 Punkte

- Beim Halbierungsverfahren (*buddy*-Verfahren) kann keine interne Fragmentierung auftreten.
- Speicherbereiche, die vor Beendigung eines Prozesses nicht mit `free` freigegeben wurden, sind bis zum Neustart des Systems unwiederbringlich verloren.
- Beim Halbierungsverfahren (*buddy*-Verfahren) kann es vorkommen, dass zwei nebeneinander liegende freie Speicherbereiche nicht miteinander verschmolzen werden können.
- best-fit* ist in jedem Fall das beste Verfahren.

d) Wie funktioniert Adressraumschutz durch Eingrenzung?

2 Punkte

- Jedes Programm bekommt zur Ladezeit mehrere Wertepaare aus Basis- und Längenregistern zugeordnet, die die Größe aller Segmente des darin laufenden Prozesses festlegen.
- Begrenzungsregister legen einen Adressbereich im physikalischen Adressraum fest, auf den alle Speicherzugriffe beschränkt werden.
- Der Lader positioniert Programme immer so im Arbeitsspeicher, dass unerlaubte Adressen mit nicht-existierenden physikalischen Speicherbereichen zusammenfallen.
- Begrenzungsregister legen einen Adressbereich im logischen Adressraum fest, auf den alle Speicherzugriffe beschränkt werden.

e) Ein laufender Prozess wird in den Zustand *blockiert* überführt. Welche Aussage passt zu diesem Vorgang?

2 Punkte

- Der Prozess terminiert.
- Es ist kein direkter Übergang von *laufend* nach *blockiert* möglich.
- Der Prozess wartet auf Daten von der Standardeingabe.
- Der bisher laufende Prozess wurde vom Betriebssystem verdrängt und ein anderer Prozess auf der CPU eingelastet.

f) Welche Aussage zum Aufbau einer Kommunikationsverbindung zwischen einem Client und Server über eine Socket-Schnittstelle ist richtig?

2 Punkte

- Der Server signalisiert durch einen Aufruf von `connect()`, dass er zur Annahme von Verbindungen bereit ist; ein Client kann dies durch `accept()` annehmen.
- Der Server erzeugt einen Socket und ruft anschließend `bind()` auf - der Client muss durch einen Aufruf von `listen()` warten, bis der Server bereit zur Annahme von Verbindungen ist.
- Nach der Erzeugung eines Sockets mittels `socket()` kann ohne weitere System- oder Funktionsaufrufe sofort eine Verbindung von einem Client durch einen Aufruf von `connect()` angenommen werden.
- Der Server richtet an einem Socket mittels `listen()` eine Warteschlange für ankommende Verbindungen ein und kann danach mit `accept()` eine konkrete Verbindung annehmen. `accept()` blockiert so lange die Warteschlange leer ist.

- g) Man unterscheidet bei Programmunterbrechungen zwischen Traps und Interrupts. Welche Aussage dazu ist richtig? 2 Punkte
- Bei der mehrfachen Ausführung eines unveränderten Programms mit gleicher Eingabe treten Traps immer an den gleichen Stellen auf.
  - Da das Betriebssystem nicht vorhersagen kann, wann ein Benutzerprogramm einen Systemaufruf absetzt, sind Systemaufrufe als Interrupts zu klassifizieren.
  - Ganzzahl-Operationen können nicht zu einem Trap führen.
  - Interrupts werden immer vom unterbrochenen Programm behandelt, Traps hingegen vom Betriebssystem.
- h) Welche der folgenden Aussagen über Einplanungsverfahren ist richtig? 2 Punkte
- Bei kooperativer Einplanung kann es zur Monopolisierung der CPU kommen.
  - Beim Einsatz präemptiver Einplanungsverfahren kann laufenden Prozessen die CPU nicht entzogen werden.
  - Probabilistische Einplanungsverfahren müssen die exakten CPU-Stoßlängen aller im System vorhandenen Prozesse kennen.
  - Asymmetrische Einplanungsverfahren können ausschließlich auf asymmetrischen Multiprozessor-Systemen zum Einsatz kommen.
- i) Welche Aussage zum Thema RAID ist richtig? 2 Punkte
- Bei RAID 0 können mehrere beteiligte Platten ausfallen, ohne dass das Gesamtsystem ausfällt.
  - Beim Einsatz von RAID 1 mit zwei Festplatten kann insgesamt genau die gleiche Datenmenge gespeichert werden, die auch ohne die Verwendung von RAID auf den selben Festplatten gespeichert werden könnte.
  - RAID 5 ist nur mit drei Platten verwendbar, da die Berechnung der Paritätsinformationen bei mehr Platten nicht möglich ist.
  - Bei RAID 4 enthält eine Platte die Paritätsinformationen, die anderen Platten enthalten Daten.

- j) Wozu dient der Maschinenbefehl *cas* (compare-and-swap)? 2 Punkte
- Um bei Monoprozessorsystemen Interrupts zu sperren.
  - Um auf einem Multiprozessorsystem einfache Modifikationen an Variablen ohne Sperren implementieren zu können.
  - Um bei der Implementierung von Schlossvariablen (Locks) aktives Warten zu vermeiden
  - Um in einem System mit Seitennummerierung (Paging) Speicherseiten in die Auslagerungspartition (*swap area*) schreiben zu können.
- k) Welche Aussage zu Seitenersetzungsstrategien ist richtig? 2 Punkte
- Bei der Seitenersetzungsstrategie *FIFO* wird immer die zuletzt eingelagerte Seite ersetzt.
  - Beim Auslagern einer Speicherseite muss der zugehörige Seitendeskriptor angepasst werden, beim Einlagern einer Seite ist das jedoch nicht nötig.
  - Bei der Seitenersetzungsstrategie *LRU* wird die Seite ersetzt, welche am längsten nicht mehr referenziert wurde.
  - Beim Einsatz der Seitenersetzungsstrategie *FIFO* kann es nicht zu Seitenflattern kommen.

**Aufgabe 1.2: Mehrfachauswahl-Fragen (8 Punkte)**

Bei den Multiple-Choice-Fragen in dieser Aufgabe sind jeweils  $m$  Aussagen angegeben,  $n$  ( $0 \leq n \leq m$ ) Aussagen davon sind richtig. Kreuzen Sie **alle richtigen** Aussagen an. Jede korrekte Antwort in einer Teilaufgabe gibt einen halben Punkt, jede falsche Antwort einen halben Minuspunkt. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, d. h. falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (~~☒~~).

Lesen Sie die Frage genau, bevor Sie antworten!

- a) Welche der folgenden Aussagen zum Thema Synchronisation sind richtig? 4 Punkte
- Die Verwendung nicht-blockierender Synchronisation benötigt besondere Unterstützung durch das Betriebssystem.
  - Ein Semaphor kann ausschließlich für mehrseitige Synchronisation verwendet werden.
  - Der Einsatz von nicht-blockierenden Synchronisationsmechanismen kann nicht zu Verklemmungen (*dead-locks*) führen.
  - Die V-Operation kann auf einem Semaphor auch von einem Faden aufgerufen werden, der zuvor keine P-Operation auf dem selben Semaphor ausgeführt hat.
  - Gibt ein Faden einen Mutex frei, den er selbst zuvor nicht angefordert hatte, stellt dies einen Programmierfehler dar; der fehlerhafte Prozess sollte dann abgebrochen werden.
  - Zur Synchronisation eines kritischen Abschnitts ist passives Warten immer besser geeignet als aktives Warten.
  - Das Sperren von Interrupts kann von Benutzerprogrammen ohne weiteres zur Synchronisation auf Multiprozessor-Systemen eingesetzt werden.
  - Durch den Einsatz von Semaphoren kann ein wechselseitiger Ausschluss erzielt werden.

- b) Welche der folgenden Aussagen zu UNIX-Dateisystemen sind richtig? 4 Punkte
- Der selbe *Inode* kann im Dateisystem im selben Verzeichnis mehrfach über verschiedene Namen referenziert werden.
  - In einem Verzeichnis darf es mehrere Einträge mit identischem Namen geben, sofern sie auf unterschiedliche *Inodes* verweisen.
  - Ein *hard link* kann nur auf Verzeichnisse, nicht jedoch auf Dateien verweisen.
  - Wird eine Datei gelöscht, so werden auch alle *symbolic links*, die auf diese Datei verweisen, gelöscht.
  - Beim lesenden Zugriff auf eine Datei über einen *symbolic link* kann ein Prozess den Fehler *Permission denied* erhalten, obwohl er das Leserecht auf dem *symbolic link* besitzt.
  - Ein *Inode* enthält u.a. den Namen der entsprechenden Datei.
  - Der *Inode* einer Datei wird getrennt von ihrem Inhalt auf der Platte gespeichert.
  - Die Anzahl der *hard links*, die auf ein Verzeichnis verweisen, hängt von der Anzahl seiner Unterverzeichnisse ab.

**Aufgabe 2: spaas (60 Punkte)**

*Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!*

Schreiben Sie ein Programm `spaas` (`sorting people as a service`), das auf dem TCP/IPv6-Port 2016 (`LISTEN_PORT`) einen Dienst anbietet, um eine von einem Client gesendete Liste von Personen sortiert an den Client zurückzugeben. Die Abarbeitung parallel eintreffender Anfragen soll von einem Arbeiter-Thread-Pool übernommen werden, dessen Größe dem Programm als einziger Kommandozeilen-Parameter übergeben wird; die Threads werden über einen entsprechend synchronisierten Ringpuffer mit Verbindungen versorgt. Damit ein fehlerhafter Client nicht zur Beendigung des gesamten Servers führt, sollen alle Threads des Servers das Signal `SIGPIPE` ignorieren.

Ein Client sendet nach erfolgreicher Verbindung eine beliebige Anzahl von Zeilen an den Server, wobei Sie davon ausgehen dürfen, dass jede Zeile nicht länger als 256 (`MAX_LINE`) Zeichen ist. Das Ende der Nutzdaten signalisiert der Client durch `EOF`. Der Server sortiert Personen alphabetisch nach ihrem Nachnamen, und bei gleichem Nachnamen nach dem Vornamen. Danach sendet er die sortierte Personenliste an den Client zurück.

Das Programm soll folgendermaßen strukturiert sein:

- Das Hauptprogramm initialisiert zunächst alle benötigten Datenstrukturen, startet die benötigte Anzahl an Arbeiter-Threads und nimmt auf einem Socket Verbindungen an. Eine erfolgreich angenommene Verbindung soll zur weiteren Verarbeitung in den Ringpuffer eingefügt werden.
- Funktion `void* threadHandler(void *)`: Hauptfunktion der Arbeiter-Threads. Entnimmt in einer Endlosschleife dem Ringpuffer eine Verbindung, und ruft zur weiteren Verarbeitung die Funktion `doWork` auf. Wird von `doWork` angezeigt, dass ein Fehler aufgetreten ist, soll eine kurze Fehlermeldung an den Client gesendet werden (z.B. "Fehler beim Bearbeiten").
- Funktion `int doWork(FILE *rx, FILE *tx)`: Liest die Zeilen, welche der Client im Format "Vorname Nachname\n" sendet, ein und baut ein dynamisch wachsendes Array aus (von Ihnen am Anfang des Programms geeignet zu definierenden) Strukturen vom Typ `struct name` auf, die den eingelesenen Vor- und Nachnamen als getrennte Elemente beinhalten. Vorname und Nachname sind dabei in der Eingabe durch mindestens ein Leerzeichen oder Tabulatorzeichen getrennt. Zeilen, die nicht genau 2 Namen enthalten, sollen vom Server ohne Fehlermeldung ignoriert werden. Die Länge des Vor- und Nachnamens soll in der Struktur auf je 127 (`MAX_NAME`) (Nutz-)Zeichen begrenzt sein; ist einer der vom Client gelesenen Namen länger als `MAX_NAME` Zeichen, so soll die entsprechende Zeile ebenfalls ignoriert werden. Nach dem Einlesen aller Zeilen sollen die Daten mittels `qsort` unter Zuhilfenahme der Vergleichsfunktion `namecmp` sortiert werden (**Hinweis**: `qsort` soll nur einmal aufgerufen werden; sowohl Vor- als auch Nachname sollen in der Vergleichsfunktion `namecmp` verglichen werden!). Zuletzt werden die sortierten Daten zeilenweise im Format ("Nachname, Vorname\n") an den Client gesendet. Tritt bei der Bearbeitung ein Fehler auf, soll die Funktion `-1` zurückgeben, ansonsten den Wert `0`.
- Funktion `int namecmp(const void *p1, const void *p2)`: Vergleichsfunktion für zwei Strukturen vom Typ `struct name`. Um die korrekte Sortierreihenfolge zu erhalten, müssen Sie in dieser Funktion den Nachnamen sowie gegebenenfalls auch den Vornamen aus den übergebenen Strukturen vergleichen.

Zusätzlich sollen Sie die Funktionen `bbPut(int value)` und `bbGet()` implementieren. Der Ringpuffer soll dabei statisch als Feld der Länge 16 (`BUFFER_SIZE`) angelegt werden. Zur Koordination stehen Ihnen Semaphore mit den Funktionen `semCreate`, `P` und `V` zur Verfügung. Die Semaphore-Funktionen müssen Sie **nicht** selbst implementieren. Die Schnittstellen entsprechen dabei dem Modul, das sie aus den Übungen kennen, und sind auf der folgenden Seite am Anfang des Programms deklariert.

Auf den folgenden Seiten finden Sie ein Gerüst für das beschriebene Programm. In den Kommentaren sind nur die wesentlichen Aufgaben der einzelnen zu ergänzenden Programmteile beschrieben, um Ihnen eine gewisse Leitlinie zu geben. Es ist überall sehr großzügig Platz gelassen, damit Sie auch weitere notwendige Anweisungen entsprechend Ihrer Programmierung einfügen können.

Einige wichtige Manual-Seiten liegen bei - es kann aber durchaus sein, dass Sie bei Ihrer Lösung nicht alle diese Funktionen oder gegebenenfalls auch weitere Funktionen benötigen.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <pthread.h>
#include <errno.h>
#include <signal.h>
#include "sem.h"

SEM *semCreate(int initVal);
void P(SEM *sem);
void V(SEM *sem);

#define LISTEN_PORT 2016
#define BUFFER_SIZE 16
#define MAX_LINE 256
#define MAX_NAME 127

static void die(const char msg[]) {
    perror(msg);
    exit(EXIT_FAILURE);
}

// Funktions- und Strukturdeklarationen, globale Variablen, etc.
```

**// Funktion main()**

**// Argumente pruefen, Initialisierungen, etc.**

**// Threads starten**

**// Socket erstellen und fuer Verbindungsannahme vorbereiten**

// Verbindungen annehmen und in den Puffer legen

// Ende Funktion main

// Funktion bbPut()

// Ende Funktion bbPut()

// Funktion bbGet()

// Ende Funktion bbGet()

M:

B:

// Funktion threadHandler

// Ende Funktion threadHandler

T:

**// Funktion doWork**



**// Zeilen vom Client einlesen, Array aufbauen**



**// Eingelesene Daten sortieren und an den Client senden**



**// Ende Funktion doWork**

**// Funktion namecmp**



**// Ende Funktion namecmp**





